Implementation and Evaluation of MPI Nonblocking Collective I/O

Sangmin Seo Robert Latham Junchao Zhang Pavan Balaji

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
{sseo, robl, jczhang, balaji}@anl.gov

***Abstract*—The well-known gap between relative CPU speeds and storage bandwidth results in the need for new strategies for managing I/O demands. In large-scale MPI applications, collective I/O has long been an effective way to achieve higher I/O rates, but it poses two constraints. First, although overlapping collective I/O and computation represents the next logical step toward a faster time to solution, MPI's existing collective I/O API provides only limited support for doing so. Second, collective routines (both for I/O and communication) impose a synchronization cost in addition to a communication cost. The upcoming MPI 3.1 standard will provide a new set of nonblocking collective I/O operations to satisfy the need of applications. We present here initial work on the implementation of MPI nonblocking collective I/O operations in the MPICH MPI library. Our implementation begins with the extended two-phase algorithm used in ROMIO's collective I/O implementation. We then utilize a state machine and the extended generalized request interface to maintain the progress of nonblocking collective I/O operations. The evaluation results indicate that our implementation performs as well as blocking collective I/O in terms of I/O bandwidth and is capable of overlapping I/O and other operations. We believe that our implementation can help users try nonblocking collective I/O operations in their applications.**

I. Introduction

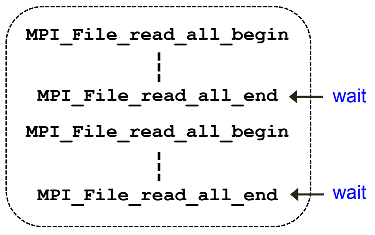
As many HPC applications deal with larger datasets, file I/O becomes more important because of its relatively slow performance. If the application requests a lot of file I/O operations, I/O time can be the main bottleneck in the application's performance. In order to alleviate the I/O performance issue, many parallel file systems and parallel I/O libraries have been introduced and widely used. The Message Passing Interface (MPI) I/O, which has been included in the MPI standard since MPI 2.0 [1], is one of these efforts that support parallel I/O operations. Moreover, many I/O optimizations—for example, nonblocking individual I/O and collective I/O—have been proposed to improve the I/O performance and to help applications developers optimize their I/O use cases.

Nonblocking operations for communication and I/O have gained much attention because they provide optimization opportunities for overlapping communication (or I/O) and computation [2]. While blocking operations do not return to the caller until they are completed, nonblocking ones initiate the operation and return immediately to the caller. After the initiation, the nonblocking operations, if possible, make their progress in the background and let the user code check the completion of the operations. If the user code can execute useful computation between the initiation and the completion of nonblocking operation, it can reduce the whole execution time by the overlapped time. Therefore, for many high-performance computing (HPC) applications that rely on heavy communication or I/O, exploiting the nonblocking operations can be critical for improving performance by hiding the communication or I/O cost.

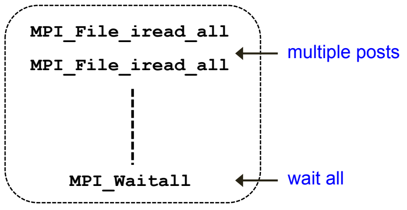
Among various nonblocking operations, nonblocking collective (NBC) I/O operations are attractive because they can take advantage of both nonblocking operations and collective operations. Applications using NBC I/O not only can overlap I/O and computation but also can leverage the optimized implementation of collective operations. Since collective I/O operations can exploit more information about I/O access patterns, they can provide the optimized performance compared with individual I/O operations. For example, with more information about file accesses, the performance can be improved by merging I/O requests of participating processes [3]. Collective operations do impose one cost: a "pseudo-synchronization" cost when early arrivals to a collective call have a data dependency on later arrivals and must wait until the laggards provide the required data [4], [5]. Nonblocking collectives allow programs to possibly hide or absorb this pseudo-synchronization cost.

The MPI standard has supported nonblocking operations for I/O and communication as immediate versions of blocking operations. The recent MPI 3.0 standard [6] provides immediate versions of all point-to-point communication and all collective communication operations. It also supports nonblocking versions for individual file I/O operations and a restricted form of NBC file I/O operations, called split collective I/O. Since the split collective I/O operations have some limitations (see Section II-A), however, the MPI Forum has sought to include one missing part–general NBC I/O operations—in the MPI standard. The upcoming MPI 3.1 standard will contain a new set of such routines.

This paper presents our initial work on the implementation of the MPI NBC I/O operations proposed for the upcoming MPI 3.1 standard and shows preliminary evaluation results. Our implementation is based on ROMIO's collective I/O algorithm [3]. It essentially replaces all blocking operations used in ROMIO's collective I/O implementation with nonblocking counterparts and uses the MPI test routines to keep track of progress or to make progress on each nonblocking operation. We use extended generalized requests [7] in order to manage the progress of NBC I/O operations. During the implementation, the original collective I/O routine is split into several



(a) Split collective I/O



(b) Nonblocking collective I/O

Fig. 1: Posting multiple collective file read operations.

code segments (i.e., different small routines) whenever any nonblocking operation is initiated. Further, the nonblocking operation issued is tested in the state machine.

The major contributions of this paper are the following.

- We review MPI NBC I/O operations and discuss their implications for applications.
- We describe a state machine–based implementation of MPI NBC I/O operations in ROMIO using the extended generalized request.
- We show the effectiveness of our implementation of MPI NBC I/O operations using benchmarks.

The rest of the paper is organized as follows. In Section II we review the limited NBC I/O features currently in MPI, and we discuss the NBC I/O operations recently added to the MPI 3.1 standard. Readers interested in our implementation of MPI NBC I/O routines will find in Section III a discussion of our state machine design and our use of extended generalized requests. In Section IV we evaluate our implementation on a Linux cluster. Section V summarizes related work. Section VI concludes the paper and presents our future work.

II. Nonblocking Collective I/O Operations

In this section, we briefly review the current nonblocking collective I/O routines for MPI, called split collective routines, and introduce a new set of general NBC I/O routines for the upcoming MPI 3.1 standard. We then discuss the implications of NBC I/O operations for applications.

A. Split Collective I/O

The current MPI standard provides I/O routines to support NBC I/O operations [6]. Since these routines divide a single collective operation into two parts, a begin routine and an end routine, they are referred to as *split collective* operations. For example, the split collective I/O routines that are equivalent to `MPI_File_read_all` are `MPI_File_read_all_begin` and `MPI_File_read_all_end`. Here, `MPI_File_read_all_begin` begins the collective read operation, and `MPI_File_read_all_end` completes the operation.

Although split collective I/O routines provide the semantics of NBC I/O operations, they have some limitations that reduce their use in applications. First, at most one active split collective operation is possible on each file handle at any time. That is, a process is not allowed to begin another split collective I/O operation on a file handle before the preceding one is completed. As shown in Figure 1(a), it has to wait for the completion of the split collective operation posted before issuing another one.

Second, split collective I/O routines do not use the `MPI_Request` handle. This limitation sometimes hinders implementers from providing efficient implementations of nonblocking operations. Since collective I/O algorithms may require more than two steps to complete a collective operation, making progress with only begin/end routines may be difficult. For example, ROMIO, a widely used MPI I/O implementation, does not provide a true immediate return implementation for split collective I/O routines [3]. It currently implements split collectives by performing all I/O in the “begin” step of the routines. ROMIO performs only a small amount of bookkeeping in the “end” step. Therefore, applications using ROMIO cannot overlap computation and I/O operations with these split collective I/O routines and cannot benefit from less restrictive synchronization.

B. Proposal for MPI 3.1 Standard

To overcome the limitations of split collective routines, the MPI Forum proposed a set of general NBC I/O routines [8]. The upcoming MPI 3.1 standard will include immediate nonblocking versions of collective I/O operations for explicit offsets and individual file pointers, and the new versions will replace the current split collective routines [9]. Specifically, the following four routines will be added.

```
MPI_File_iread_all(..., MPI_Request *req)
MPI_File_iwrite_all(..., MPI_Request *req)
MPI_File_iread_at_all(..., MPI_Request *req)
MPI_File_iwrite_at_all(..., MPI_Request *req)
```

The NBC I/O routines have the same interface as their blocking counterpart except that the last parameter is `MPI_Request` instead of `MPI_Status`. These routines initiate a collective read or write operation and return a request handle. The returned request handle can be tested or waited on for completion with `MPI_Test` or `MPI_Wait`, respectively, or any of their variants. Unlike split collective I/O operations, multiple NBC I/O operations can be posted on a single file handle without waiting for the completion of preceding operations. For an NBC I/O routine for individual file pointer (i.e., `MPI_File_iread_all` or `MPI_File_iwrite_all`), the file pointer is advanced appropriately when the routine returns, so that the following I/O operations on the file pointer operate on the right offset.

C. Implications for Applications

With the proposed NBC I/O operations, applications can take advantage of benefits of both collective I/O operations and

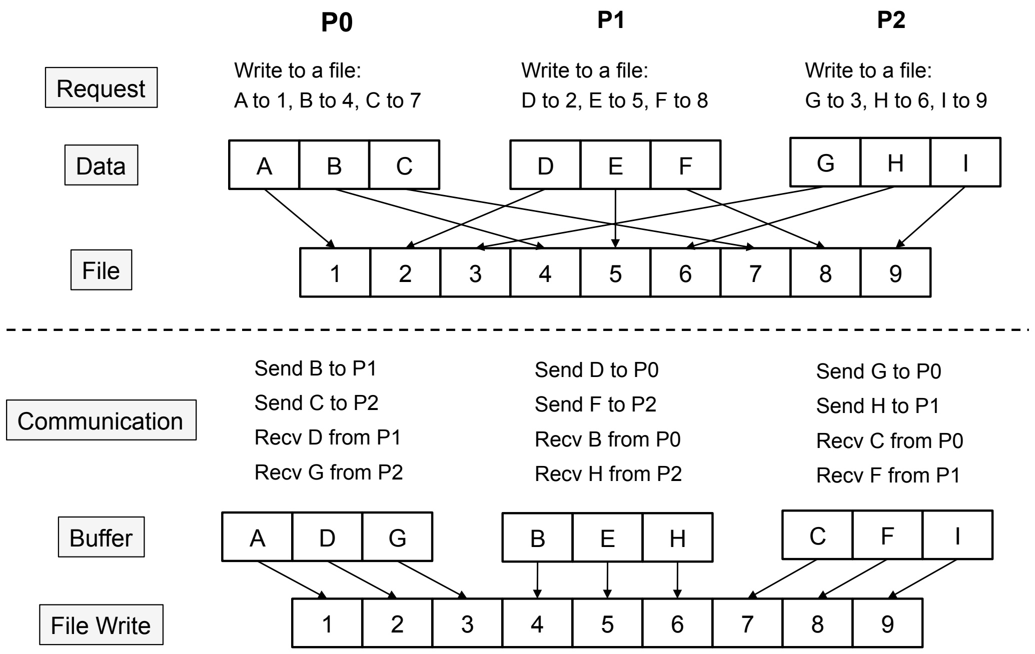


Fig. 2: Example of collective write operation in ROMIO.

nonblocking operations. Basically, NBC I/O operations can deliver optimized performance just as blocking I/O operations can because they take the same parameter information as their blocking counterpart. A high-quality implementation of NBC I/O operations can give applications I/O performance similar to that with corresponding blocking I/O operations. Applications can also benefit from overlapping I/O operations and other computation or communication by issuing nonblocking operations.

In addition, new NBC I/O operations enable different collective I/O operations to be overlapped. For instance, users can post multiple NBC I/O operations on a single file handle and wait for the completion of all operations using `MPI_Waitall`, as shown in Figure 1(b). Note that split collective I/O operations do not allow the posting of more than one operation on a single file handle at a time (Figure 1(a)).

III. Implementation of NBC I/O Operations

This section describes our implementation of NBC I/O operations. Specifically, we implement the four NBC I/O routines introduced in Section II-B inside ROMIO [3].

A. *Collective I/O in ROMIO*

Collective I/O operations in ROMIO were implemented by using a generalized version of the extended two-phase method presented in [10], [3], so that any noncontiguous I/O requests can be handled. Here we briefly summarize that two-phase method, in order to give background for our NBC I/O implementation.

The two-phase I/O method basically splits a collective I/O operation into two phases. For example, in the first phase for the write operation, each process sends its noncontiguous data, from the point of view of file, to other processes in order for each process to rearrange the data for a large contiguous region in a file. Then, each process writes a big contiguous region of a file with collected data instead of accessing many small noncontiguous regions. This two-phase method can combine a large number of noncontiguous requests into a small number of contiguous I/O operations and thus can improve performance significantly. This is possible because collective I/O interfaces provide the MPI I/O implementation with information about I/O requests of all processes. For the read operation, processes read data from a file first and then communicate with each other to distribute data to processes in order to satisfy the original read request.

Figure 2 shows an example of collective write operations handled in ROMIO. In the figure, three processes (P0, P1, and P2) request to write three blocks from their memory to noncontiguous regions in a file. P0 requests to write the data block A to the file block 1, B to 4, and C to 7. Similarly, P1 requests to write D to 2, E to 5, and F to 8. P2 also requests the same pattern of write operations. If we handle requests of all processes independently, each process may need three individual write operations because file offsets for the write operation are not contiguous. For example, P0 may have to access three file offsets (1, 4, and 7) to write three data blocks (A, B, and C).

Since this is a collective write request, however, ROMIO can optimize the requests of all processes by exploiting the information about file offsets and data distribution among processes. First, all processes exchange their request information (e.g., offsets, lengths) and determine which blocks can make contiguous regions in a file. Then, each process distributes its own blocks to other processes so that each process can have contiguous data to write to a file. In Figure 2, P0 sends block

B and C to P1 and P2, respectively, and receives block D and G from P1 and P2, respectively. P1 and P2 also, similarly to P0, communicate data blocks with other processes. After this communication phase, in their buffer P0 has A, D, and G; P1 has B, E, and H; and P2 has C, F, and I, respectively. All processes then can write their buffer to a contiguous region in the file with a single operation. For example, P0 writes its buffer consisting of A, D, and G to the contiguous region, from 1 to 3, in the file.

B. State Machine-Based Implementation

Our implementation of NBC I/O operations uses the same general algorithm as the blocking collective I/O operations in ROMIO, but we replace all blocking communication or I/O operations with nonblocking counterparts and divide the original routine into two separate routines when the blocking operation is changed. In addition, we use request handles to make progress or keep track of progress.

Figure 3 illustrates how we convert original blocking operations in MPI_File_write_all to nonblocking ones and split MPI_File_write_all into different routines in order to implement MPI_File_iwrite_all. First, we define a struct, struct nbcio_status, that is used internally to keep track of the status of NBC I/O operation. For example, in Figure 3(b), struct nbcio_status has a request handle nio_req for the ongoing NBC I/O operation, a request handle cur_req for the current nonblocking communication or I/O operation in progress during the NBC I/O, a state variable state to record the status of operation, and other fields. A struct nbcio_status variable is allocated inside MPI_File_iwrite_all and is deallocated when the request handle nio_req is released (not shown in Figure 3).

Second, we change all blocking calls (e.g., MPI_Alltoall and ADIO_WriteStrided) in MPI_File_write_all of Figure 3(a) to nonblocking counterpart calls (e.g., MPI_Ialltoall and ADIO_IwriteStrided) and use cur_req of struct nbcio_status to obtain the related request handle. The cur_req handle is used to check the progress of MPI_Ialltoall and ADIO_IwriteStrided in the state machine code of Figure 4. We also insert code that saves the current state of NBC I/O operation right after each nonblocking routine call.

We then split the modified routine into a set of different routines according to locations where we put nonblocking routine calls. The result of the code split is shown in Figure 3(b). The original routine MPI_File_write_all is divided into three routines: MPI_File_iwrite_all, iwrite_all_fileop, and iwrite_all_fini. MPI_File_iwrite_all is an entry routine that initiates a collective file write operation. It includes the code from the beginning of MPI_File_write_all to MPI_Ialltoall. It also contains code that allocates a struct nbcio_status variable and creates a generalized request class, gen_class, followed by allocating a request object using the extended generalized request interface (MPIX_Grequest_class_create and MPIX_Grequest_class_allocate) [7]. The request handle returned to the user can be used to test or wait for the completion of this collective file write operation.

```
int MPI_File_write_all(..., MPI_Status *status) {
  ...
  MPI_Alltoall(...);
  ...
  ADIO_WriteStrided(...);
  ...
}
```

(a) MPI_File_write_all

```
struct nbcio_status {
  MPI_Request nio_req; /* for this NBC I/O op. */
  MPI_Request cur_req; /* current nonblocking op. */
  nbcio_state state;   /* current state */
  ...                  /* other fields */
};

MPIX_Grequest_class greq_class = 0;

int MPI_File_iwrite_all(..., MPI_Request *req) {
  ...
  nio_status = malloc(sizeof(struct nbcio_status));
  if (greq_class == 0)
    MPIX_Grequest_class_create(..., iwrite_all_poll_fn,
                               ..., &greq_class);
  MPIX_Grequest_class_allocate(greq_class, nio_status,
                               &nio_status->nio_req);
  memcpy(req, &nio_status->nio_req, sizeof(MPI_Request));

  /* use nonblocking communication */
  MPI_Ialltoall(..., &nio_status->cur_req);
  nio_status->state = IWRITE_ALL_STATE_COMM;
  return error_code;
}

void iwrite_all_fileop(struct nbcio_status *nio_status) {
  ...
  ADIO_IwriteStrided(..., &nio_status->cur_req);
  nio_status->state = IWRITE_ALL_STATE_FILEOP;
}

void iwrite_all_fini(struct nbcio_status *nio_status) {
  ...
  nio_status->state = IWRITE_ALL_STATE_COMPLETE;
  MPI_Grequest_complete(nio_status->nio_req);
}
```

(b) MPI_File_iwrite_all

Fig. 3: Example of implementing MPI_File_iwrite_all from MPI_File_write_all by replacing blocking operations with nonblocking ones.

The second routine iwrite_all_fileop includes the code after MPI_Alltoall in Figure 3(a) and ends with the ADIO_IwriteStrided call. This routine is invoked by iwrite_all_poll_fn in Figure 4 only when the communication in the first routine is completed.

The third routine iwrite_all_fini executes the remaining code and completes the request for MPI_File_iwrite_all. It is called when the nonblocking file write operation in iwrite_all_fileop is completed.

We manage the progress of NBC I/O operations by using a state machine. Since our implementation of an NBC I/O routine consists of several routines and they must be executed according to the order of original two-phase algorithm, we need to keep track of which routine is executing at a certain point. We solve this issue by maintaining a state machine for each NBC I/O operation. Figure 4 shows an example of the state machine that manages the progress of NBC I/O

```
int iwrite_all_poll_fn(void *extra_state, ...) {
  ...
  int flag = 0;
  nio_status = (struct nbcio_status *)extra_state;
  switch (nio_status->state) {
    case IWRITE_ALL_STATE_COMM:
      MPI_Test(&nio_status->cur_req, &flag, status);
      if (flag) iwrite_all_fileop(nio_status);
      break;
    case IWRITE_ALL_STATE_FILEOP:
      MPI_Test(&nio_status->cur_req, &flag, status);
      if (flag) iwrite_all_fini(nio_status);
      break;
    case IWRITE_ALL_STATE_COMPLETE:
      break;
    ...
  }
  ...
}
```

Fig. 4: Example of state machine code for the implementation of `MPI_File_iwrite_all` in Figure 3.

operations. In the figure, when the state of the NBC I/O operation is `IWRITE_ALL_STATE_COMM`, the request handle related with this state is tested to check whether the previous communication is completed. If it is completed, the next routine, here `iwrite_all_fileop`, is called. Otherwise, the state machine code returns to the caller, and the caller again checks the status of operation. The state machine terminates when its state reaches the final state, namely, `IWRITE_ALL_STATE_COMPLETE`, and the request handle for the entire NBC I/O operation is marked as completed.

Since ROMIO uses the extended generalized request interface to implement nonblocking independent I/O operations [7], we also exploit it to implement NBC I/O operations and to easily integrate new routines into ROMIO. The extended generalized requests add poll and wait routines to the standard MPI generalized requests. These routines enable users to utilize the test and wait routines of MPI in order to check progress on or make progress on user-defined nonblocking operations. For example, the `iwrite_all_poll_fn` routine in Figure 4 is registered when a generalized request class is created (See the `MPI_File_iwrite_all` routine in Figure 3(b)) and is called when the test routine is called with the request handle returned from the NBC I/O routine. On the other hand, standard generalized requests are unable to make progress with the test or wait routines: their progress must occur completely outside the underlying MPI implementation (typically via pthreads or signal handlers).

C. *Progress of NBC I/O Operations*

The MPI standard specifies that all nonblocking calls are local and return immediately irrespective of the status of other processes [6]. This will also be the case for NBC I/O routines in the upcoming MPI 3.1 standard [9]. The NBC I/O routine initiates the I/O operation and returns a request handle, which must be passed to a completion call. The implementation of NBC I/O operations may make progress implicitly or may require the user's code to call `MPI_Test` or `MPI_Wait` in order to make explicit progress.

Our current implementation of NBC I/O operations provides neither asynchronous nor implicit progress. The user has to call the test or wait routines to explicitly make progress of NBC I/O operations. Although this explicit progress management seems to be a burden to users, it is currently a common practice in implementing nonblocking operations [5], [11]. We can exploit progress threads in order to support asynchronous progress for NBC I/O operations, but we leave this task for future work.

TABLE I: Target platform

Configuration	Blues
Number of nodes	310
Processor	Intel Xeon E5-2670
Clock frequency	2.60 GHz
Cores per node	16
HW threads per core	2 (disabled)
Memory per node	64 GB
File system	GPFS
Interconnect	QLogic QDR infiniband
Topology	Fat-tree
OS	CentOS 6.6
Compiler	GCC 4.4.7

IV. Evaluation

This section presents the evaluation methodology and initial evaluation results of our implementation of NBC I/O operations.

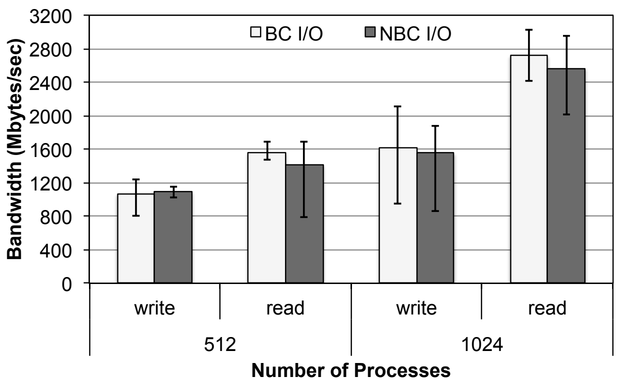
A. *Methodology*

Target platform. For the evaluation, we use the Blues cluster at Argonne National Laboratory. Blues consists of 310 compute nodes, and a GPFS file system is shared among all nodes on the cluster. Each compute node has two Intel Xeon E5-2670 CPUs and supports 16 cores, but currently hyperthreading is disabled on all nodes. Table I shows details about the target platform.

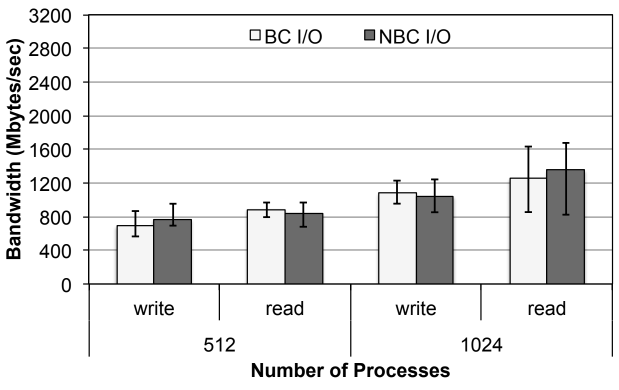
MPI implementation. We implemented the NBC I/O routines introduced in Section II-B inside ROMIO. Our current NBC I/O implementation is integrated into MPICH 3.2a2 [12] as `MPIX` routines (e.g., `MPIX_File_iwrite_all` instead of `MPI_File_iwrite_all`) because new NBC I/O routines are not currently included in the MPI standard. We note that ROMIO is distributed as part of MPICH. Our implementation (described in Section III) uses the extended generalized request provided by MPICH.

Benchmarks. To evaluate our NBC I/O implementation, we use the `coll_perf` benchmark in the ROMIO test suite and its modifications in order to use NBC I/O operations or to overlap collective I/O operations and computation. We also use a microbenchmark to measure the performance of overlapping multiple I/O operations.

Using these benchmarks, we compare our NBC I/O routines with their blocking counterpart routines in terms of file write and read bandwidth and overlapping with other operations. Since the implementation of split collective I/O routines in ROMIO is the same as that of blocking collective I/O routines, we do not include comparison results with split collective I/O routines.



(a) ppn=8



(b) ppn=16

Fig. 5: File write and read bandwidth with 512 and 1,024 processes.

B. I/O Bandwidth

We measure the file I/O bandwidth with the `coll_perf` benchmark for blocking collective I/O operations. For NBC I/O operations, we modify `coll_perf` by replacing blocking I/O routines with their corresponding NBC I/O routines followed by wait routines. For example, `MPI_File_write_all` is converted to a pair of `MPI_File_iwrite_all` and `MPI_Wait`. The `coll_perf` benchmark measures the I/O bandwidth for writing and reading a 3D block-distributed array, which is created by `MPI_Type_create_darray`, to a file. It sets the file view with the 3D block-distributed array, and thus the file access pattern is noncontiguous. We use a 3D block-distributed array whose global size is $2176 \times 1152 \times 1408$ integers (about 14 GB).

Figure 5 shows results of file I/O bandwidth measurement. We use 512 and 1,024 processes on Blues with different processes per node (ppn) settings. Figure 5(a) and Figure 5(b) illustrate results when ppn is 8 and 16, respectively. Note that each execution with ppn=8 uses twice as many nodes as that with ppn=16. In Figure 5, BC I/O and NBC I/O denote results of blocking collective I/O and nonblocking collective I/O operations, respectively. Each execution is done six times for each configuration. Figure 5 shows the average bandwidth and maximum and minimum bandwidths measured.

The results indicate that our implementation of NBC I/O operations achieves, on average, bandwidth similar to that of blocking collective I/O operations. The reason is that our NBC I/O implementation is based on the same algorithm of blocking collective I/O in ROMIO, and the benchmark invokes the wait routine right after issuing an NBC I/O routine. In other words, the benchmark for NBC I/O does almost the same amount of work as that for BC I/O. The NBC I/O routines ideally should have more overhead only from additional function calls and memory management. The similar bandwidth results in Figure 5 indicate that our approach is efficient and does not cause significant overhead compared with blocking I/O operations. Where the results of NBC I/O show better bandwidth than those of BC I/O, we believe that they come from jitter in I/O operations or communication. The better bandwidth results in Figure 5(a) than those in Figure 5(b) are because I/O traffic is distributed to more nodes in the case of ppn=8. Again, we note that the same number of processes with ppn=8 use more nodes than those with ppn=16.

```
MPI_File_write_all();
Computation();
```
(a) Blocking I/O with computation

```
MPI_File_iwrite_all(..., &req);
for (...) {
  Small_Computation();
  MPI_Test(&req, &flag, ...);
  if (flag) break;
}
Remaining_Computation();
MPI_Wait(&req, ...);
```
(b) NBC I/O with computation

Fig. 6: Executing I/O operation with computation.

C. Overlapping I/O and Computation

Basically, one cannot overlap blocking I/O and other computation, and thus they should be executed successively as shown in Figure 6(a). However, computation code can be added between the NBC I/O routine and the wait routine in order to overlap I/O and computation. We insert into the `coll_perf` benchmark some synthetic computation code like that in Figure 6(b). We regularly call `MPI_Test` to explicitly make progress, as explained in Section III-C. Consequently, the big computation, `Computation()`, in Figure 6(a) is divided into small pieces of computation, `Small_Computation()` and `Remaining_Computation()`, in Figure 6(b). In order to ensure fair comparison, the amount of both computations is kept the same. We measure the execution time of each case in Figure 6.

Figure 7 shows performance results of adding some computation to the `coll_perf` benchmark. The benchmark uses the global array size, $1536 \times 1024 \times 1024$ integers (6 GB) and 64 processes with ppn=8 for the evaluation. In the figure, BC I/O and NBC I/O represent the execution times of blocking collective I/O with computation and NBC I/O with computation, respectively. We ran each case five times; Figure 7 presents the average execution time. The portion of dark gray in each bar denotes the execution time of computation; light gray means the I/O time, which is calculated by subtracting the computation time from the entire execution time.

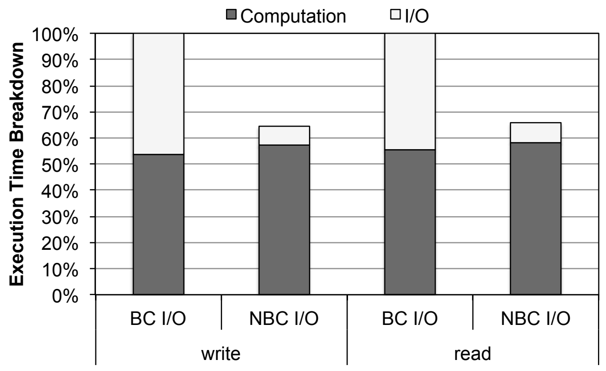


Fig. 7: Performance results of overlapping I/O and computation.

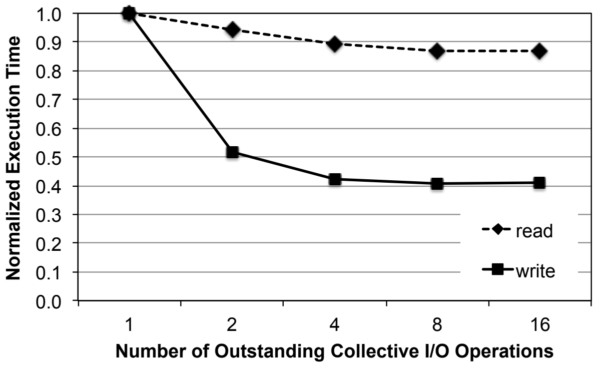


Fig. 8: Performance results of overlapping multiple I/O operations.

NBC I/O in Figure 7 shows that 84% of the I/O time is overlapped with computation for the write operation and 83% for the read operation. Consequently, the entire execution time of NBC I/O is reduced by 36% for write and 34% for read, respectively, compared with that of BC I/O. The slight increase in the computation time of NBC I/O in Figure 7 seems to come from noises in the measurement or cache effect resulting from switching between computation code and the test routine. We note, however, that the workload of computation and I/O is the same for both BC I/O and NBC I/O.

If the I/O operation is completely overlapped with computation, the I/O time will be fully hidden by the computation time. This is not the case for our NBC I/O implementation, however, mainly because of the explicit progress requirement. To make progress on the NBC I/O operation, we have to call the test routine many times, but doing so incurs some call overhead. Furthermore, it is difficult to know how frequently the test routine should be called in order to hide the progress of the NBC I/O operation with computation while minimizing the call overhead. If the test routine is called after one state in the state machine of Figure 4 is completed, it delays the progress of the I/O operation. On the other hand, if it is called while one state in the state machine is ongoing, it causes the call overhead. For this experiment, we frequently invoke the test routine in order not to delay the I/O operation. This problem could be mitigated if we used asynchronous progress threads. We will investigate this issue in our future work.

D. Overlapping Multiple I/O Operations

New NBC I/O operations overcome the limitation of split collective I/O that does not allow more than one outstanding collective I/O operation (Section II). With the new NBC I/O routines, we can initiate multiple collective I/O operations at a time and wait for the completion of all posted operations, as shown in Figure 1(b).

To see the benefit of issuing multiple collective I/O operations, we run a microbenchmark that performs 16 collective I/O operations with different file offsets. Each collective I/O operation accesses 1 GB of a file, and thus the benchmark writes or reads 16 GB of the file in total. For the experiment, 64 processes with ppn=8 are used. We measure the execution time varying the number of outstanding collective I/O operations.

Figure 8 illustrates the execution times with different numbers of outstanding collective I/O operations. Execution times are normalized to that of 1, the case when only one collective I/O operation is posted at a time. The results show that issuing multiple collective I/O operations at a time reduces the execution time up to 59% with eight outstanding collective I/O operations for write and 13% with 16 outstanding collective I/O operations for read, respectively. These results indicate that multiple collective I/O operations can be overlapped by using NBC I/O routines, although the overlapping depends on the progress engine of the MPI implementation. Here, we show that our implementation is capable of overlapping multiple collective I/O operations. If applications exploit NBC I/O operations in this way, they can obtain another performance benefit that cannot be achieved by using blocking collective I/O or split collective I/O operations.

V. Related Work

To our best knowledge, only one paper has been published about the design and evaluation of NBC I/O operations. Venkatesan et al. [11] proposed general NBC I/O operations that became the basis of the proposal for MPI 3.1 standard described in Section II-B. The researchers implemented NBC I/O operations in the Open MPI I/O library (OMPIO) [13] using the libNBC library [5]. Their implementation leverages the concept of collective operations schedule used in the libNBC library, whereas our implementation exploits the state machine and the extended generalized request to keep track of progress of NBC I/O operations. In addition, their implementation requires modification of the progress engine of the underlying libNBC, whereas our implementation does not need to modify the progress engine of MPI implementation if it provides the extended generalized request interface. We plan to compare the performance and efficiency of the two implementations.

On the other hand, collective I/O operations have been studied by many researchers. The two-phase method, which is widely used in collective I/O implementations, was originally proposed in [14] for accessing distributed arrays from files. Thakur and Choudhary [10] extended the basic two-phase algorithm for accessing sections of out-of-core arrays. Then, Thakur et al. [3] described ROMIO's implementation of collective I/O based on the extended two-phase method. Our

NBC I/O implementation is based on the algorithms presented in that latter work.

Some variants and optimizations have been proposed in order to improve the performance of two-phase methods. Chaarawi et al. [15] presented segmentation algorithms derived from ROMIO's collective I/O method, and Sehrish et al. [16] proposed a multibuffer pipelining optimization that overlaps the request aggregation phase and the I/O phase.

Furthermore, researchers have investigated various directions in collective I/O. Blas et al. [17] proposed view-based collective I/O, which is a file system-independent collective I/O optimization based on file views. Coloma et al. [18] presented their MPI collective I/O implementation to provide better flexibility for tuning, better research platform, and easier maintenance than ROMIO's implementation. For application developers to easily utilize collective I/O, Yu et al. [19] developed the transparent collective I/O library that can reduce programming effort by providing POSIX-like interfaces.

Since our approach can be applied to other collective I/O algorithms, previous research on collective I/O is complementary to our work. It is beyond the scope of this paper, however, to compare different collective I/O algorithms or optimizations because we do not propose a new algorithm or optimization techniques here.

VI. Conclusions and Future Work

This work presents an implementation of NBC I/O operations, which will be part of the upcoming MPI 3.1 standard. Our implementation is based on the extended two-phase algorithm used in collective I/O implementation in ROMIO and utilizes the state machine and the extended generalized request to maintain progress of NBC I/O operations. The evaluation results indicate that our implementation performs as well as blocking collective I/O in terms of I/O bandwidth and is capable of overlapping I/O and other operations. We believe that our implementation can help users try NBC I/O operations in their applications.

We plan to work on asynchronous progress of NBC I/O operations in order to overcome the shortcomings of the explicit progress requirement. We also plan to apply NBC I/O operations to real applications in order to see the benefit of NBC I/O and further improvements required for them. Additionally, we will compare our approach with other approaches, for example, the work in [11].

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," November 2003.

[2] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04, June 2004, pp. 298–305.

[3] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99, February 1999, pp. 182–189.

[4] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006, pp. 1–12.

[5] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, November 2007, pp. 52:1–52:10.

[6] Message Passing Interface Forum, "MPI: A Message-Passing Interface standard version 3.0," September 2012.

[7] R. Latham, W. Gropp, R. Ross, and R. Thakur, "Extending the MPI-2 generalized request interface," in *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. PVM/MPI '07, September 2007, pp. 223–232.

[8] "Add immediate versions of nonblocking collective I/O routines," https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/273.

[9] Message Passing Interface Forum, "MPI 3.1 standardization effort," http://meetings.mpi-forum.org/MPI_3.1_main_page.php.

[10] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, December 1996.

[11] V. Venkatesan, M. Chaarawi, E. Gabriel, and T. Hoefler, "Design and evaluation of nonblocking collective I/O operations," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI '11, September 2011, pp. 90–98.

[12] "MPICH," http://www.mpich.org/.

[13] M. Chaarawi, E. Gabriel, R. Keller, R. L. Graham, G. Bosilca, and J. J. Dongarra, "OMPIO: A modular software architecture for MPI I/O," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI '11, September 2011, pp. 81–89.

[14] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *SIGARCH Comput. Archit. News*, vol. 21, no. 5, pp. 31–38, December 1993.

[15] M. Chaarawi, S. Chandok, and E. Gabriel, "Performance evaluation of collective write algorithms in MPI I/O," in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS '09, May 2009, pp. 185–194.

[16] S. Sehrish, S. W. Son, W.-k. Liao, A. Choudhary, and K. Schuchardt, "Improving collective I/O performance by pipelining request aggregation and file access," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, September 2013, pp. 37–42.

[17] J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero, "View-based collective I/O for MPI-IO," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '08, May 2008, pp. 409–416.

[18] K. Coloma, A. Ching, A. Choudhary, W. keng Liao, R. Ross, R. Thakur, and L. Ward, "A new flexible MPI collective I/O implementation," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, ser. Cluster '06, September 2006, pp. 1–10.

[19] Y. Yu, J. Wu, Z. Lan, D. H. Rudd, N. Y. Gnedin, and A. Kravtsov, "A transparent collective I/O implementation," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, May 2013, pp. 297–307.